計算機アーキテクチャ 77－12
（１９８９．７．１４）

細粒度並列実行支援機構

松 本 尚

日本アイ・ビー・エム株式会社　東京基礎研究所

近年，VLSI技術の大きな進歩により，多数のプロセッサを用いたマルチプロセッサシステムの実現が可能となり，そのためのアーキテクチャやアルゴリズムの研究が盛んに行なわれている。マルチプロセッサシステムでは従来から指摘されているように，プロセッサ間のデータの通信の問題やそれに伴う通信競合の問題そして同期の問題が深刻になる。特に大きな並列度を得ようとして各プロセッサに割り振るタスクの粒度を細かくするときには，これらの問題によるオーバー・ヘッドが十分に小さくなくては効率を上げることはできない。本稿では，マルチプロセッサ上での細粒度並列実行を支援する機構の満たすべき条件の検討を行ない，その条件を満たすオーバー・ヘッドが無視できる簡易な同期機構と共有バスの特徴を生かし通信競合を緩和する簡易なスヌープ・キャッシュ制御機構とを提案する。

Fine Grain Support Mechanisms

Takashi Matsumoto

IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo 102

VLSI technology makes multiprocessor systems feasible recently. However, overhead caused by inter-processor communication and synchronization still prevent efficient execution of parallel programs. Especially, reduction of those overhead is mandatory in systems which focus large scale and fine grain parallelism. This paper discusses requirements for mechanisms supporting fine grain parallelism efficiently and describes two simple mechanisms matching those requirements; one is a synchronization mechanism with low overhead, and the other is a snoop-cache control mechanism reducing overhead accompanied with cache consistency.

1.はじめに

現存のマルチプロセッサ上で細粒度(数命令程度)の並列実行を行わせようとすると，プロセッサ間の同期や通信のオーバー・ヘッドのためにプロセッサ一台で走らせるのが最も効率が良いといった事態がよく起こる。これらのオーバー・ヘッドを十分に小さくし，細粒度の並列性をうまく実行することができれば，従来プロセッサ一台で実行せざるおえなかった処理が並列に実行できる可能性が増える。また，大きな粒度の並列度に比べて細粒度の並列度は自動的に並列化コンパイラやパラレライザによって抽出し易い。この面からも効率の良い細粒度並列実行は重要である。

細粒度の並列性を効率良く実行するためのアーキテクチャとして，VLIWマシン[1]やデータ・フロー・マシン[2]が有名であり，これらでは 1命令または 1演算を単位として並列度を抽出している。しかし，これらは従来型の単一プロセッサのアーキテクチャとかなり異なっており，性能の良い計算機を実現するには大きな努力を要する。また，簡単に大きな粒度の並列度が取りだせる問題に対しては従来型の単一プロセッサを使用した既存のマルチプロセッサ・システムで十分である。そこで，既存のマルチプロセッサ・システム(密結合型)に同期やデータ通信を効率化する機構を追加することによって，細粒度の並列実行を可能にする。本稿ではそのための機構の条件を検討し，二つの機構を提案する。

第2章で細粒度並列実行を可能にするハードウェア資源の管理方針を述べ，第3章でオーバー・ヘッドの無視できる同期機構を述べ，第4章で共有バス型のマルチプロセッサのデータ通信量を改善しバス競合を緩和するスヌープ・キャッシュ制御機構を述べ，第5章でまとめを述べる。本稿で述べる機構は筆者の提唱する細粒度並列実行支援マルチプロセッサ（FGSM: Fine Grain Support Multiprocessor）[3]の一部として考案された。

2.ハードウェア資源の管理方針

今日までの単一プロセッサ型の計算機システムはハードウェア資源をなるべく仮想化する方向で発展してきた。しかし，細粒度の並列実行を効率よく行うためには仮想化の追究ばかりではうまくいかない。仮想化のレベルやOSの機能といったものも含めて見直さなければ，細粒度の並列実行を効率良く実現するマルチプロセッサ・システムを設計することはできない。

まず，混乱を防ぐために用語を統一する。ジョブは一つの独立した処理単位を表わし，ジョブ間ではOSを介してのみ同期を取ったりデータのやりとりができる。プロセスはOSがプロセッサのスケジューリングを行う基本単位で，プロセスが複数集まって一つのジョブを形成する。同じジョブに属するプロセス間の同期や通信はOSまたは共有メモリを介して行なわれる。また，一つのジョブの中のプロセスの実行順序は指定可能である。シュレッド(shred)は単一のプロセッサに割り付けられる処理単位を表わし，複数のシュレッドが集まって一つのプロセスを形成する。タスクは細粒度の処理単位を表わし，タスクが複数集まってプロセスを形成し，その中の一つのプロセッサに対応するタスクの集まりがシュレッドである。同じプロセス内で異なるシュレッドに属するタスク間の同期や通信は支援ハードウェアがあればそれを直に制御して行うことも許される。

マルチ・ユーザーやマルチ・ジョブといった機能が実現できる範囲でハードウェアの管理をユーザー・サイドに行わせオーバー・ヘッドを減らすことを基本方針とする。具体的には，OSはスケジューリングの際，一つのプロセスに指定された台数（シュレッドの数）のプロセッサを同時に割当てる機能を持つ。シュレッドとプロセッサが一対一に対応し，同じプロセスに属するシュレッドは同時にプロセッサに割り付けられる。また，保護についてはプロセス・レベル以上で保護を行ない，同じプロセスに属するシュレッド間の特別な保護機構は考えない。ハードウェア資源の管理方針についての詳細は文献(3)を参照されたい。

3.軽い同期機構

3.1.待ち合わせ同期

細粒度のタスクを異なるプロセッサに割り振って，その間で同期を取っても効率が落ちないようにすることが軽い同期機構の目的である。ここで同期といっているのはタスク間の実行順序関係つまりタスク間の依存関係を守るための待ち合わせの問題である。一般に同期問題は相互排除問題等の様々な問題を含むが，細粒度並列実行ではタスク間の依存関係を守るための待ち合わせが一番頻繁に起こり一番重要な問題である。タスク間で相互排除が必要な時は，静的にコンパイラまたはプログラマがそのタスクの実行順序を決定してしまうことでタスク間の実行順序関係に帰結できる。ここで提案する同期機構は同一プロセスに属し異なるプロセッサに割り付けられるタスク間で待ち合わせを行うためのものである。

3.2.軽い同期の必要性

従来型の単一プロセッサを用いたマルチプロセッサの場合，たとえ全プロセッサが一つのクロックに基づいて動作していても，キャッシュの状態や通信の競合による実行の遅れといった実行時以前には予測不可能な要因があるため同期は不可欠である。従来からマルチプロセッサで行なわれている比較的軽い同期法は共有メモリ上の共有変数を用いる方法である。この方法はコストの掛かる共有メモリへのアクセスを伴い，メモリを浪費しないために同じ共有変数を異なる時点での同期に流用しようとすると排他的に共有変数にアクセスする必要があり，さらにコストが掛かる。そこで，いくつかのマルチプロセッサ・システムでは，同期のための共有変数をアクセス・コストの安い専用共有レジスタに取るようにしたシステム[4] [5]や，同期のためにデータ通信路が頻繁に排他的に使われるのを嫌って

同期専用の通信路を設けたシステム[6]などがある。確かに，これらで同期のコストは改善されるが，共有変数を使った同期のための処理と共有変数へのアクセスの直列化によるコストが無くなるわけではない。共有変数を使った同期のための処理だけでも数命令程度掛かるので，従来の方法では数命令規模の細粒度並列実行の同期は行えない。

3.3.軽い同期機構の基本アイデア

同期機構を具体的に述べる前に，この機構の基本となっているアイデアを列挙する。

<1> 同期待ち時にコンテクスト・スイッチ（ここではシュレッド・スイッチ）等を行ってプロセッサの制御権をOSに返したりすると，細粒度のタスク間の同期を扱っているため大きなオーバー・ヘッドになる。また，同期の待ち合わせが成立しなければ，タスクの依存関係等を保存するためシュレッド内の先の命令を実行できない。以上の2点より，待ち合わせが成立するまでプロセッサを一時的にWAIT状態にして止めればよいことが判る。これをハードウェア的に実現すれば待ち合わせのために同期変数をチェックする処理のオーバー・ヘッドはなくなる。

<2> 並列化コンパイラまたはプログラマが同期の必要な位置を指示するので，同期位置の情報はプロセッサへの命令列（シュレッド）の中に埋め込まれ，プロセッサが命令列を実行中に埋め込まれた情報を解釈し，適宜同期の必要性を外部に出力するのが望ましい。何故なら，命令列とは別に同期を取るべき位置の情報があると仮定すると，その情報を同期機構にセットするオーバー・ヘッドが掛かりハードウェアも複雑になる。

<3> 同期が頻繁に起こってもプロセッサ間のデータ通信を妨げないように，プロセッサ間に同期情報専用の通信路を設ける。また，プロセッサが異なれば基本的には同期情報の通信競合が起こらないようにする。

<4> 同じプロセスに属するシュレッド（プロセッサ）でも同期位置によって同期を必要とする組合せが変化する。その様子を図1に示す。図1の同期位置1ではシュレッド2を除いたすべてのシュレッドが待ち合わせを行う必要があることを示している。もちろん，同期位置毎に同期を取るシュレッドの組合せを示す情報を持っていれば指定されたシュレッド間だけで同期をとれる。しかし，この方法ではハードウェアが複雑になり現実的でない。そこで，本来は待ち合わせの必要のない同期位置であっても，他に同期を必要とするシュレッドがある位置にはシュレッドにダミーとして同期を要求する同期情報（図1内の黒丸印）を挿入する。これにより毎回一つのプロセスに属する全プロセッサが待ち合わせを行えばよく，ハードウェアが大幅に簡略化できる。

<5> ハードウェアを工夫してダミーの同期要求の挿入位置が少々ずれてもオーバー・ヘッドが生じないようにする。

<6> 図1では全プロセッサが揃うまで待ち合わせを行う同期が示されているが，実際の細粒度のタスクの同期ではタスクの順序関係（典型的には生産者と消費者の関係）を保存するための同期が主である。順序関係上，先に実行が終了すべきタスクが実際実行時に先に終了した場合，他の

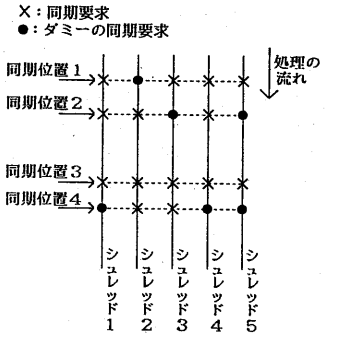


図1 同期位置とダミーの同期要求

シュレッドに属するそれに続く関係のタスクの開始を待つ必要はなく，待つと逆にオーバー・ヘッドになる。このオーバー・ヘッドをハードウェアを工夫し回避する。

<7> プロセッサ資源の仮想化と有効利用を支援するため，OSによるプロセッサのグループ分けを可能にするハードウェアを設ける。

3.4.軽い同期機構の構成

図2に軽い同期機構を含むシステムの全体構成を示す。ここでは便宜的にプロセッサ間の結合方式は共有バス方式とする。3.3.節の<3>を満たすため同期情報を伝達する同期信号バスを別に設け，各プロセッサをそのバスの一本のラインに対応させる。つまり，プロセッサ台数分のラインからなる同期信号バスを設ける。プロセッサ毎に同期コントローラが設けられ，これが同期信号バスを使って同期の成立を検出する。

3.3.節の<2>の考え方を生かすため，プロセッサの命令コードに同期情報のためのフィールドまたはタグを新設するか，同期情報のためのプリフィックス命令を新設す

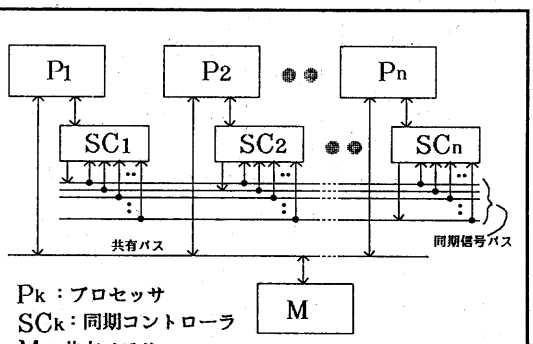


図2 同期機構を含むシステム

る。3.3.節の＜4＞ の考え方からこの同期情報は『この情報が付加された命令の直前で待ち合わせを行う必要がある』（直前ではなく直後でもかまわないが，便宜上直前に決める）という同期要求を示す 1ビットの情報で良いことが判る。しかし，＜5＞ と ＜6＞ の考え方を実現するため本論文では 2ビットつまり 4種の情報とする。プロセッサはこの同期情報をそれが付加された命令の実行直前にSSIG0 (Synchronization SIGnal), SSIG1 の信号線を通して同期コントローラに対して出力する。 4種の同期情報の動作と意味を以下に示す。

NONE プロセッサは同期に関し何もしない。(00)

RREQ プロセッサは同期情報を出力後， 同期コントローラから SACK 信号が返るまで命令実行中断状態になる。(Real REQuest : 01)

APRV プロセッサは同期情報を出力後，命令実行を続ける。意味は次回の同期成立の承認。
(APpRoVal : 10)

PREQ プロセッサは同期情報を出力後，命令実行を続ける。意味は次回の同期成立の先取り。
(PreREQuest : 11)

同期情報が RREQ の時のみプロセッサは同期コントローラから SACK (Synchronization ACKnowledge) 信号が返るまで実行中断状態になる。これは 3.3.節の ＜1＞ の考え方に基づいている。 OS によるプロセスのスケジューリングを保証するため，プロセッサは同期の待ち合わせによる実行中断状態でも受付け可能な割込機能を持つ必要がある。

図3に同期コントローラの構成を示す。同期コントローラは次の二つの動作を基本にしている。

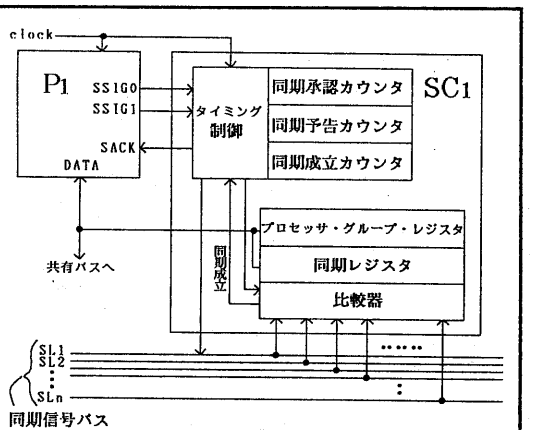


図3 同期コントローラの構成

(1) 自分の担当するプロセッサが SSIG 信号線上に RREQ の同期情報を出力した時に，同期信号バスの自分に割当てられたラインをアクティブにする。
(2) 同期信号バスを監視して，同期レジスタに登録されたプロセッサ群に対応する同期信号バスの全てのラインがアクティブになると SACK をプロセッサに出力し，同期信号バスの自分のラインをネゲイトする。

上記の2つの動作のみでは3.3.節の ＜5＞ と ＜6＞ の考え方が実現できない。そこで，同期コントローラ内部に以下の 3つのカウンタを設け，カウンタの状態を用いて動作を拡張する。なお今後，同期レジスタに登録されたプロセッサ群に対応する同期信号バスの全てのラインがアクティブになることを同期条件成立と表現する。

同期承認カウンタ 同期条件未成立のAPRV の数を計数する。

同期予告カウンタ RREQの前に挿入され同期条件未成立の PREQ の数を計数する。

同期成立カウンタ RREQの前に挿入され同期条件が成立した PREQ の数を計数する。

同期コントローラの拡張された動作は以下の通りである。なお，カウンタの初期値は0 である。

{1} RREQがプロセッサから入力されると，同期成立カウンタが 0 でなければ同期成立カウンタの値を 1減らして SACK をプロセッサにすぐに返し，同期成立カウンタが 0 で同期予告カウンタが 0 でなければ何もせず，同期成立カウンタも同期予告カウンタも0であれば同期信号バスの自分に割当てられたラインをアクティブにする。
{2} APRV がプロセッサから入力されると，同期承認カウンタを 1増やし，自分の同期信号ラインがインアクティブならアクティブにする。
{3} PREQ がプロセッサから入力されると，同期予告カウンタを 1増やし，自分の同期信号ラインがインアクティブならアクティブにする。
{4} 同期信号バスを監視して，同期条件の成立を検出すると，

- **{4-1}** 同期承認カウンタが0 でないとき，同期承認カウンタを 1減らす。その結果，同期承認カウンタと同期予告カウンタが共に 0 になれば自分の同期信号ラインをネゲイトし，一方でも 0 でなければ再びアクティブにする。
- **{4-2}** 同期承認カウンタが0 で同期予告カウンタが0 でないとき，同期予告カウンタを 1減らし同期成立カウンタを1増やす。その結果，同期予告カウンタが0になれば自分の同期信号ラインをネゲイトし，0でなければ再びアクティブにする。
- **{4-3}** 同期承認カウンタが0で同期予告カウンタが0 のとき，同期成立カウンタを 1増やし，自分の同期信号ラインをネゲイトする。

{5} プロセッサが RREQ を信号線に出して SACK 待ちで実行中断状態の時に，同期成立カウンタが0 でなくなったら，すぐに SACK をプロセッサに出力し同期成立カウンタを 1減らす。

なお，図3 内のプロセッサ・グループ・レジスタは同期レジスタに登録できるプロセッサを制限するためのレジスタで，　3.3.節の <7> の考え方を反映している。このレジスタは OS レベルでしか変更できず，同じプロセスに属するシュレッドに対応するすべてのプロセッサが登録されている。同期レジスタも通常は同じプロセスに属するすべてのプロセッサが　OS　によって登録されているが，ユーザー・レベルで変更可能であり，プロセッサ・グループ・レジスタに登録されているプロセッサなら同期レジスタの登録から削除したり再登録したりできる。プロセッサ・グループ・レジスタと同期レジスタはプロセッサ台数分のビット幅を持ったレジスタとしてインプリメントされる。OS によってプロセスが一時実行中止されプロセッサ資源が他のプロセスに解放される際は，同期コントローラ内のカウンタやレジスタはプロセッサ内のレジスタと同様に保存される必要がある。

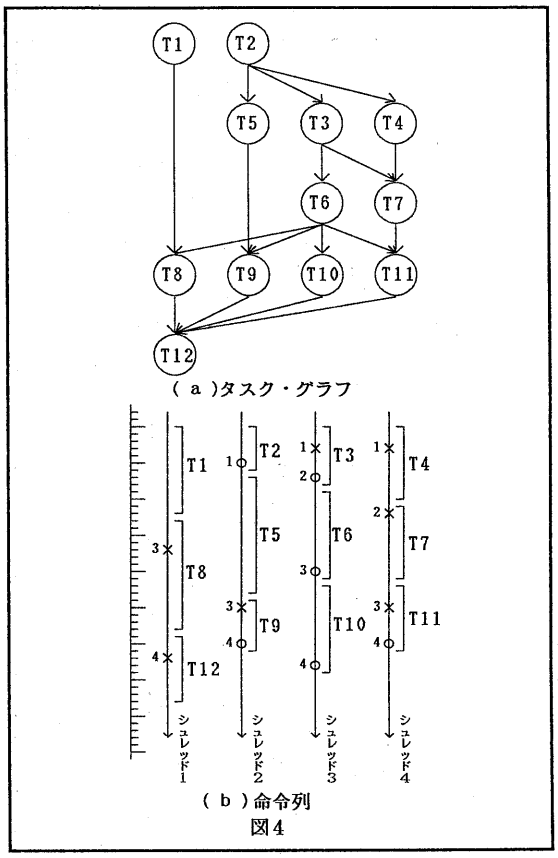
また，この同期機構が命令の実行順序を保証するため，同期コントローラには以下の制約がある。プロセッサから同期情報が入力されてもプロセッサがそれ以前に実行した命令に伴うデータ通信が通信路の遅延等で完了していないときは，同期コントローラはその通信が完了するまで同期信号バス上の自分のラインをインアクティブからアクティブに変更する動作は行わない。

3.5.軽い同期機構の使用法

この節では例を用いて，軽い同期機構の使用法を説明する。同期コントローラ内部の三種のカウンタと PREQ, APRV の同期情報の存在意義を明確にするため，3.4.節で (1) (2) として述べた基本動作のみの同期コントローラを使った場合と，3.4.節の {1} {2} {3} {4} {5} で述べた動作を行う同期コントローラを使った場合との比較を行ないながら説明を行う。

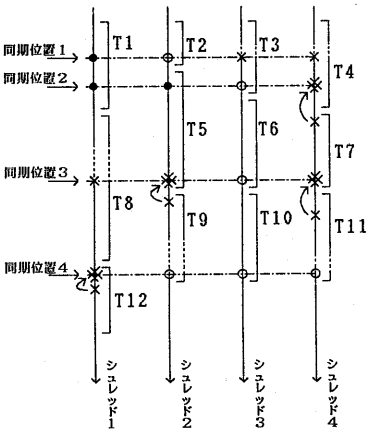
例として図4(a) のタスク・グラフで表わされる処理を考える。ノードが細粒度のタスクを表わし，矢印（有向枝）がタスク間の依存関係を表わしている。図4(b) は図4(a) の処理を4つのシュレッドに割り振ったときの命令列の模式図で長さが命令量を表わし，簡単のため同期待ちがなければ実行時間も長さに比例するものとする。図中の丸印とバツ印はシュレッド間の依存関係が問題となる命令の命令列内の位置を表わし，バツ印の位置の命令は同じ番号の付いた丸印の位置の命令より先には実行できないことを示している。具体的には，丸印の一つ前の命令で変数に計算結果を代入し，バツ印での命令ではその結果を参照するといった依存を表わしている。

まず，基本動作のみの同期コントローラを用いた場合の図4 の処理に対する同期の取り方を説明する。通常すべてのシュレッドは同時に待ち合わせを行うので，図5(b) から4回の待ち合わせのために 4つの RREQ を各シュレッドに挿入する必要があることが判る。その挿入位置を決定するために同期を含んだ実行時間を見積もると図5(a) のようになる。破線部分は同期待ちのためプロセッサが実行中断状態にあることを示している。その結果，黒丸印はダミーで同期を取るべき位置を表わし，二重バツ印は本来のバツ印の代りに待ち合わせのオーバー・ヘッドを減らす見積上最良の地点を表わしている。結局，シュレッド毎の命令列に戻ると，図5(b) の二重丸印の位置の命令に RREQの情報を付加することになる。しかし，データ通信路の競合による実行の遅延やキャッシュのミスヒット等の実行前に見積もれない要因で，黒丸印や二重バツ印の位置が最良の RREQ 挿入位置でなくなる可能性がある。また，黒丸印や二重バツ印の位置の決定に手間も掛かる。

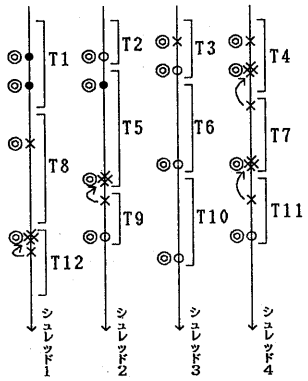


（ａ）タスク・グラフ
（ｂ）命令列
図4

次に，カウンタを含んだ動作を行う同期コントローラを用いた場合の図4　の処理に対する同期の取り方を説明する。ダミーの同期要求（黒丸印）や図5(a) の同期位置4 のように同期を承認（丸印）するシュレッドが複数ある場合の同期情報挿入位置の決定の手間と同期のオーバー・ヘッドの緩和に APRV と同期承認カウンタが用いられる。ダミーの同期要求を出したり同期を承認（丸印）する側のシュレッドは同期条件が成立するのを待つ必要はなく，命令の実行を続けられる。そこで，ダミーの同期要求（黒丸印）に関しては，同期条件の成立順序が変わらない範囲



(a)実行時間の見積



(b) RREQ 挿入位置

図5

でシュレッドの前方に APRV を挿入する。丸印の同期の承認に関しては，その位置に RREQ ではなく APRV を挿入する。同期承認カウンタが通過してまだ同期条件が成立していない APRV の数を記憶しており，その値を用いて同期コントローラがプロセッサと独立して同期信号バスのコントロールを行う。図5(a) で同期の被承認（バツ印）側に対する最適な RREQ の挿入位置として二重バツ印の位置を求めた。この位置決定の手間と実行時間のずれに対するオーバー・ヘッドの増加を抑えるために PREQ と同期予告カウンタと同期成立カウンタが用いられる。バツ印はこれから先の命令を行ってもよいという承認を受ける側である。本質的に同期条件の成立順序が変わらない範囲で，バツ印の前のどこで同期条件が成立してもかまわない。そこで，バツ印の RREQ 1つにつき 1つ PREQ をその前方に挿入し，PREQ と RREQ で挟まれた区間ではどこでも同期条件が成立可能にする。自由度を増すために，PREQ の挿入位置が他のバツ印を越えてシュレッドの前方に挿入可能にする。同期予告カウンタは同期条件未成立の通過した PREQ の数を計数し，同期条件が成立すると同期予告カウンタは 1 減らされ，同期成立カウンタが 1 増やされる。 RREQ にプロセッサが出会うと，同期成立カウンタが 1以上なら同期成立カウンタを 1 減らして命令実行を続け， 0 なら同期条件が成立するまで実行を中断する。これらの拡張により図5(b) の黒丸印や二重バツ印に関してオーバー・ヘッドを出さずに挿入できる位置に幅がでて，データ通信路の競合等の実行の遅れを吸収できる可能性が増える。

図4(b)より4種の同期情報の挿入位置を決定したものが，図6 である。丸印は APRV の三角印は PREQ のバツ印は RREQ の挿入位置を示している。当然，何も印のない場所は NONE である。挿入位置の決定法は，まず待ち合わせ同期の順序を見積もる（この例では明らか）。図4(b) で元々，丸印の地点は APRV のバツ印の地点は RREQ の挿入位置になる。次に，ダミーの同期を行うための APRV の挿入位置を決定する。シュレッド1 は最初の 2回の同期を行う必要がないので，ダミーとしてシュレッドの頭に 2つの APRV を挿入する。シュレッド2 では 2回目の同期が必要ないので， ダミーとして 1回目の同期の APRV の直後に APRV を挿入する。最後に，PREQ の挿入位置を決定する。PREQ は RREQ に対応しており，PREQ と RREQ で囲まれた部分のどこでも待ち合わせが成立可能となるので，PREQ と RREQで囲まれる範囲を大きくする方がオーバー・ヘッドを減らす点では有利である。PREQ の挿入位置は APRV 挿入位置（図6 の丸印）を越えない範囲で命令列の前方に移動できる。可能な限り前方に移動した結果が図6 の三角印の地点であり，PREQ の最良の挿入地点である。

4.スヌープ・キャッシュ制御機構

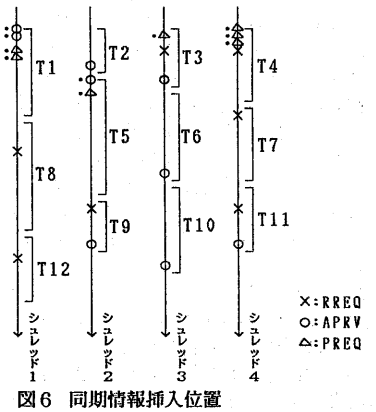


図6 同期情報挿入位置

4.1.共有バス共有メモリ型マルチプロセッサ

プロセッサ間のデータ通信を従来型のプロセッサで頻繁に行う場合は，共有メモリを介しての通信が適している。そこで，共有メモリとプロセッサとの間の結合方式を検討する必要がある。最もハードウェア構造が単純で経済的な結合方式として共有バス方式があり，バス上でのデータ通信が競合しなければ共有メモリへのアクセス・コストも低い。しかし，プロセッサ数が増加するとバス競合が起こりデータ通信のオーバー・ヘッドが増大する。このバス競合を大幅に緩和する方法としてスヌープ・キャッシュ[7]が考案され，現在多くのシステムで実用化されている。共有メモリとプロセッサとの間の結合方式として共有バス方式を採用し，プロセッサ毎にスヌープ・キャッシュを設けることにする。そして，この章ではスヌープ・キャッシュを制御してさらにバス競合を緩和する機構について述べる。

4.2.キャッシュ・プロトコール

共有バス共有メモリ型でスヌープ・キャッシュを持ったマルチプロセッサ・システムにおいて，プロセッサ間の共有データの取扱い方（プロトコール）が共有バスの使用頻度を決定し，システムのパフォーマンスに大きな影響を及ぼす。そして，従来から invalidate タイプ，update タイプのプロトコール等，複数のタイプのプロトコールが知られている。しかし，どのプロトコールにも一長一短があり，すべてのタイプの共有データに対して適しているわけではない。細粒度並列実行を考えるとプロセッサ間のデータの受渡しが頻繁に起こり，update タイプのプロトコールは非常に都合が良い。しかし，ページング等でローカル変数が共有データになる可能性もあり，また大粒度並列実行の中には invalidate タイプの方が効率がでるものが報告されている[8]ので，update タイプのプロトコールにプロトコールを固定するのが良いとは言えない。

4.3.動的なプロトコールの切替の必要性

メモリ毎（変数やワークエリア毎）に適したキャッシュのプロトコールが異なっている。また，細粒度並列実行ではプロセッサ間のデータの受渡つまりバス・トラフィックが頻繁に起こる。よって，メモリ毎に動的にプロトコールを切替制御することがバス・トラフィックを減らし，パフォーマンスを向上する上で重要である。しかし，従来のシステムの多くはプロトコールが一種類に固定されている。複数のプロトコールの混在を許すシステム[9]もあるが，そのシステムはプロトコールをプロセッサ毎に選択するようになっている。同じメモリに対してプロセッサ毎にプロトコールを変更する利点はあまりない。バス・トラフィックを減らすことができれば，共有バスを飽和させずに接続可能なプロセッサの数が増え，システム全体の性能が高まる。

4.4.新しいプロトコール

効率的にプロトコールを切替る機構を実現することが可能となると，ある種のデータにのみ有効だが，他のデータに用いると効率が悪くなるため今まで使われなかったようなプロトコールも有効に使えるようになる。そのようなプロトコールに対してはその適合するデータにのみ選択的に用いられるように制御する。このようなプロトコールの例として鈴木[10]によって提案された all-read というプロトコールを説明する。このプロトコールはバス・スヌープを行っているキャッシュが他のキャッシュによる共有バスを使っての read の際に積極的にバス上のデータを取り込むプロトコールである。但し，データを取り込む際にキャッシュ内のデータの書戻しのための共有バスの使用を必要とする場合は取り込まないものとする。このプロトコールは以下のような場合に適している。全プロセッサが同じデータを参照する必要があるときに各プロセッサが個別にリードを行ない，メモリからキャッシュにデータを張り付けるとすると，キャッシュ内にデータが持ってこられるまでに，データ数×プロセッサ数 のバス・トラフィックが共有バスで発生する。そこで， all-readをこのデータに対するプロトコールとして用いれば，データ数だけのバス・トラフィックで済み効率が良い。

4.5.スヌープ・キャッシュ制御機構の構成

動的なプロトコールの切替のために，アドレス毎にプロトコールのタイプを表す情報を付加したのでは，その情報のための記憶領域や管理のハードウェアがかなりの量となり，効率的ではない。そこで一定の記憶領域ごとにプロトコールのタイプを表す情報を付加し管理する。ここでは実現の容易さからページ管理機構に注目し，ページ毎にその情報を付加することにする。図7 に本機構のページ・エントリーの例を示す。図7 のページ・エントリーの P1, P0 はアクセス権を示すビットである。 A, D, E は仮想記憶の管理に用いられるビットで，それぞれページ内がアクセスされたか，書き替えられたか，ページが実メモリに存在するかを示す。そして，T2, T1, T0 が本発明の特徴であるページのプロトコールのタイプを表すビットである。つまり，このページに属するデータはどのタイプのプロトコールを使ってアクセスするかを示す情報である。プロセッサ（又は MMU）がメモリ・アクセスする際に，このアクセスはどのプロトコールで処理しなければならないかを示すため，このプロトコールのタイプを示すビットを外部に出力する信号線をプロセッサに設ける。そして，図8 のようにマルチプロセッサを構成し，キャッシュから共有バス上にアクセスが出るとき，つまり共有バスのトラフィックが必要なときには，共有バスにもプロトコールの

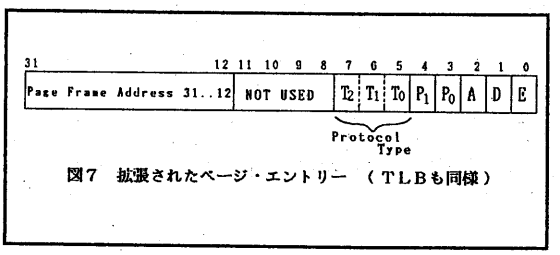


図7 拡張されたページ・エントリー （TLBも同様）

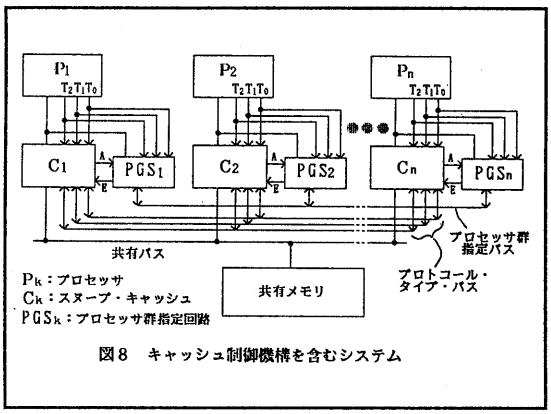


図8　キャッシュ制御機構を含むシステム

タイプを示す信号線が出力され，他のキャッシュはこの信号線でプロトコールを選択しながらバス・スヌープ（バス監視）を行う。

コンパイラによるアクセス・パターンの解析やプログラマによるプロトコール・タイプの指示等によって，変数やワークエリアをそれに適したプロトコールのタイプを持つページ内に割り振る。

all-readタイプに類するプロトコールでは，データを取り込んで欲しいキャッシュ群（プロセッサ群）を指定できることが望ましい。この機能の実現のためには，システム・バス上でキャッシュ群を指定する情報（グループ識別番号）を伝えるためのバスとその情報を蓄えておいて all-read タイプのアクセス時に外部に出力するようなハードウェアがあればよい。そして，各キャッシュはスヌープ動作で，バスのアクセスが all-read タイプの際に，プロセッサ群を指定するバスも監視して自分のプロセッサが選択されているときのみデータ取り込みの動作を行うようにする。図8 のプロセッサ群指定回路とプロセッサ群指定バスがこの処理を実現する部分である。このバスのバス幅は $\log_2(n+2)-1$本（nはプロセッサ台数）である。プロセッサ群の指定は OS がプロセスにプロセッサを割り付ける毎に設定を行えばよい。

5.おわりに

従来型の単一プロセッサを用いた密結合マルチプロセッサ上で細粒度並列実行を可能とするためのハードウェア資源の管理方針と同期機構とデータ通信機構について考察を行った。そして，オーバー・ヘッドの無視できる同期機構と共有バス上のバス・トラフィックを減少させるスヌープ・キャッシュ制御機構を提案した。なお，並列化コンパイラを介したこれらの機構の使用法については別途詳細に報告する予定である。今後の課題としてはマルチプロセッサ上での細粒度並列実行を支援する他の機構の検討，数命令程度以上の粒度をもった並列度を抽出する新手法の考案，細粒度支援機構の中粒度以上での使用法の検討などが挙げられる。

謝辞

本研究をサポートして下さった黒川利明氏，同期機構に関し御討論いただいた中田武男氏と福田宗弘氏，研究の機会と助言を与えて下さった鈴木則久所長に感謝いたします。

文献

1. R.P.Colwell et al.: A VLIW Architecture for a Trace Scheduling Compiler. *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (October 1987) pp. 180-192.
2. T.Shimada et al.: An Architecture of a Data Flow Machine and Its Evaluation. *Proc. COMPCON84 Spring* IEEE (1984) pp. 486-490.
3. 松本: 細粒度並列実行支援マルチプロセッサの検討. 並列処理に関する「指宿」ミニシンポジウム, 信学会CPSY (August 1989).
4. C.D.Polychronopoulos: Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design. *IEEE Trans. Comput., vol.37, no.8* (August 1988) pp. 991-1004.
5. T.J.Teixeira and R.F.Gurwitz: Stellix:UNIX for a Graphics Supercomputer. *Proc. of the Summer 1988 USENIX Conf.* (June 1988) pp. 321-330.
6. B.Beck et al.: VLSI Assist for a Multiprocessor. *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (October 1987) pp. 10-20.
7. M.Papamarcos and J.Patel: A low-overhead coherence solution for multiprocessors with private cache memories. *Proc. 11th Annual IEEE Int. Symp. on Computer Architecture* (1984) pp. 348-354.
8. S.J.Eggers and R.H.Katz: A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. *Proc. 15th Annual Int. Symp. on Computer Architecture* (May 1988) pp. 373-382.
9. 森脇,清水: 高速並列処理ワークステーション（TOP-1）. 第38回情処全国大会論文集 (1989) pp. 1456-1457.
10. 鈴木則久: Private Communication (August 1988).